



# C++ STL Concurrencia

# Concurrencia

- ❖ Ejecución de múltiples tareas simultáneamente.
- ❖ Permite mejorar el throughput o responsividad de un programa.
- ❖ Soporte en STL a través de threads.
- ❖ Soporte de concurrencia a nivel de system-level, soportado por hardware y sistemas operativos modernos.
- ❖ Soporte de múltiples threads ejecutándose simultáneamente en un único address space.
- ❖ Soporte en C++ de “memory model” y operaciones atómicas.

# Concurrencia - Threads

- ❖ Task: cálculos que pueden ser ejecutados simultáneamente con otros.
- ❖ Thread: representación en system-level de un task en un program.
- ❖ Representado por `std::thread` en `<thread>`.

```
void f(); // function
struct F { // function object
    void operator()();
};
void user() {
    std::thread t1{f}; // se ejecuta f() en un thread separado
    std::thread t2{F{}}; // se ejecuta F()() en un thread separado
    t1.join(); // wait for t1
    t2.join(); // wait for t2
}
```



# Concurrencia - Sincronización

- ❖ Los threads comparten el address space.
- ❖ Comunicación entre threads por objetos compartidos.
- ❖ Problemas de acceso sin control a objetos compartidos. Data races.
- ❖ Mecanismos de sincronización de acceso. Locking.

```
void f() {  
    cout << "Hello ";  
}
```

```
struct F {  
    void operator() () {  
        cout << "Parallel World!";  
    }  
};
```

- ❖ Ambas utilizan el objeto `cout` sin sincronización.
- ❖ Posible salida por consola:

**PaHeralll111el o World!**

# Threads - Argumentos

```
void f(const vector<double> &v);

struct F {
    vector<double> &v;
    F(vector<double> &vv) : v{vv} { }
    void operator()();
};

int main() {
    vector<double> some_vec {1,2,3,4,5,6,7,8,9};
    vector<double> vec2 {10,11,12,13,14};

    thread t1 {f, some_vec};
    thread t2 {F{vec2}};

    t1.join(); // si se llama al destructor antes que el thread
    t2.join(); // termine su procesamiento se llama a std::terminate
}
```

# Threads - join

- ❖ No devuelve el control hasta que el thread complete su procesamiento

```
void tick(int n) {  
    for (int i = 0; i != n; ++i) {  
        this_thread::sleep_for(second{1});  
        output("Alive!");  
    }  
}  
  
int main() {  
    thread timer{tick, 10};  
    timer.join();  
}
```

# Threads - detach

- ❖ Dejar que un thread se ejecute más allá de su destructor es un mal error.
- ❖ Si uno desea que un system thread sobreviva a su thread (handle), hay que utilizar detach.

```
void run2() {  
    thread t{tick, 10};  
    t.detach(); //let tick run independently  
}
```

# Threads - namespace `this_thread`


## ❖ Define operaciones sobre el thread activo

- `get_id()`                      retorna el *ID* del current thread
- `yield()`                        da al scheduler la oportunidad de correr otro thread
- `sleep_until(tp)`            pone el current thread a dormir hasta el timepoint `tp`
- `sleep_for(d)`                pone el current thread a dormir durante la duracion `d`



# Threads - Sincronización

```
mutex m;    // controlling mutex
int sh;     // shared data

void f() {
    unique_lock<mutex> lck{m};    // acquire mutex 
    sh += 7;                     // manipulate shared data
}                                // release mutex implicitly
```

- ❖ Responsabilidad del programador saber qué mutex protege qué datos compartidos y tomarlo antes de operar sobre los datos.

```
class Record {
public:
    mutex rm;
    // ..
};
```

# Threads - Sincronización

- ❖ Es común acceder a múltiples recursos para realizar cierta acción. Puede provocar deadlocks.
- ❖ Supongamos que el thread1 toma el mutex1 y luego trata de adquirir el mutex2 mientras el thread2 toma el mutex2 y trata de adquirir el mutex1.

```
void f() {  
    unique_lock<mutex> lck1 {m1, defer_lock}; // no adquiere  
    unique_lock<mutex> lck2 {m2, defer_lock}; // inmediatamente  
    unique_lock<mutex> lck3 {m3, defer_lock}; // el mutex  
  
    lock(lck1, lck2, lck3);  
    // ...  
}; // release implícito
```

→ [Dining philosophers problem](#)

# Threads - Eventos

- ❖ Es común que un thread espere por un cierto evento externo a él.
- ❖ El evento más simple es esperar el paso de un cierto tiempo.

```
using namespace std::chrono;  
  
auto t0 = high_resolution_clock::now();  
this_thread::sleep_for(milliseconds{20});  
auto t1 = high_resolution_clock::now();  
cout << duration_cast<nanoseconds>(t1-t0).count() << "ns\n";
```



- ❖ Funciones de manejo de tiempo en <chrono>

# Sincronización - Eventos

- ❖ Una `condition_variable` es el mecanismo que permite un thread esperar por una determinada condición (satisfecha por otro thread).
- ❖ Supongamos 2 threads comunicándose pasándose mensajes por una cola. Necesitamos sincronización entre quien produce y quien consume mensajes de la cola.

```
class Message {  
    // Objeto a ser comunicado  
};  
  
queue<Message> mqueue;  
condition_variable mcond;  
mutex mmutex;
```

# Sincronización - Eventos

```
void consumer() {
    while( true ) {
        unique_lock<mutex> lck{mmutex};           // acquire mutex
        while( mqueue.size() == 0)
            mcond.wait(lck);                       // release lock and wait
        auto m = mqueue.front(); mqueue.pop()      // get message
        lck.unlock();                               // release lck
        // process m ...
    }
}

void producer() {
    while( true ) {
        Message m;
        // fill message
        unique_lock<mutex> lck{mmutex};           // protect operation
        mqueue.push(m);
        mcond.notify_one();                       // notify
        // release lock
    }
}
```

# Concurrencia – Nivel conceptual

- ❖ Mecanismos en STL para permitir operaciones concurrentes a nivel conceptual, en vez de directamente manejar threads y locks.
- ❖ En el header `<future>`
- ❖ `future` y `promise`: para retornar un valor (y excepción) de una tarea ejecutada en un thread distinto.
- ❖ `packaged_task` para facilitar la utilización de `future` y `promise`.
- ❖ `async()` para ejecutar tareas de una manera similar a un llamado a función.